Digital Phase Locked Loop (PLL): Matlab Part

Objective

In this assignment, you will

- Design a simple digital PLL with a single-pole loop filter
- Simulate the response of the PLL in MATLAB

Pre-Lab

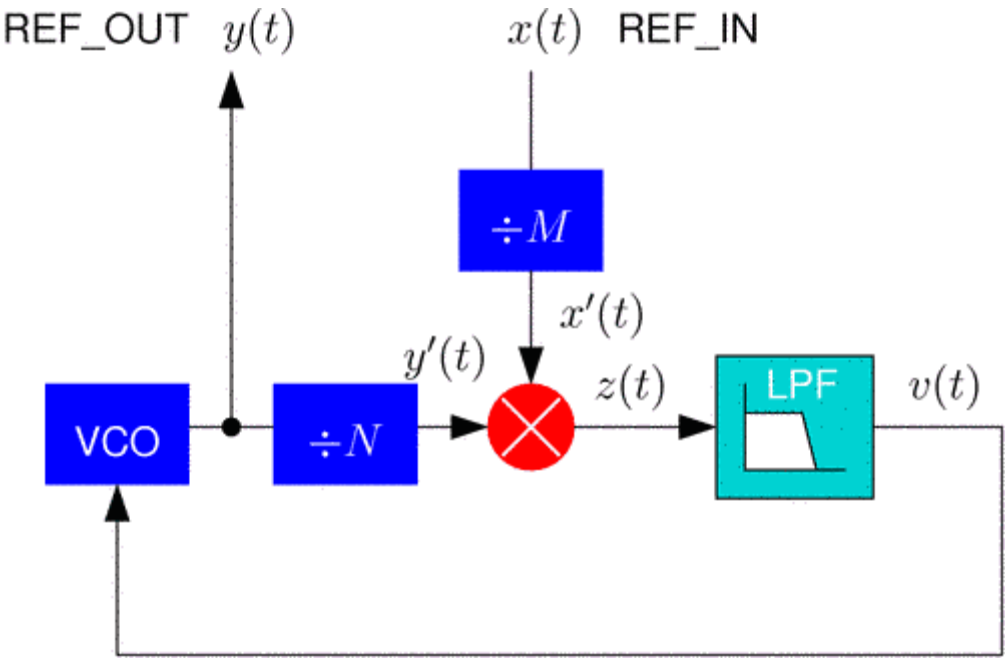
Please read the background and answer the questions at the bottom under “Pre-Lab Exercise” below. We will ask to see your answers at the beginning of the lab and may ask you to explain any of your answers.

Background

Synchronization plays a major role in many signal processing applications. For example, in a communications system, the receiver must synchronize its local oscillator and matched filter sample points correctly with respect to the transmitted signal in order to properly decode the information stream. In the Simulink labs, I gave you a pre-designed block that implemented the PLL, but in this lab, you will learn some details about the internal operation of the PLL and code and simulate it in MATLAB.

Basic PLL Operation

The diagram below shows the block diagram of the basic PLL to be implemented in this assignment:



The usual function required of the PLL is to take an incoming reference sinusoid *x*(*t*) and generate an output sinusoid *y*(*t*) whose phase closely tracks the phase of the input. Depending on the setup, the PLL can also generate a signal that is a specified rational multiple of the input reference frequency, as well as shift the phase by a prescribed amount. In communications systems, PLLs are used for two important synchronization tasks: (1) carrier recovery, which involves synchronizing the local oscillator (LO) to the incoming signal and (2) symbol timing recovery, or properly aligning the sample times at the matched filter output.

The operation of the PLL can be understood intuitively as follows. Taking the product of the local reference *y*(*t*) from the VCO and the input reference *x*(*t*) produces a signal with two components: a low frequency component that is proportional to the phase difference of *y*(*t*) and*x*(*t*) and a doubled frequency component. The low pass filter (LPF) removes the doubled frequency, just leaving the slowly varying voltage*v*(*t*) that is proportional to the phase difference. Feeding this phase difference to the voltage controlled oscillator (VCO) causes the oscillator to speed up when *y*(*t*) is lagging *x*(*t*) or to slow down when *y*(*t*) is leading *x*(*t*). When the phases reach proper alignment, the control voltage to the VCO will be close to 0, which means the loop is *tracking*.

In the following sections, we provide some details about the operation of the PLL, allowing you to code a simple software implementation. We should note that we are really implementing an analog PLL digitally on the DSP. There are actually *all-digital* PLL implementations that use only binary arithmetic (not floating-point multiplies), which can be more efficient and well-suited for certain applications, such as clock synchronization.

The Linearized Second-Order PLL

Assuming that the input signal is

$$x(t) = \cos[2\pi f t + \phi(t)]$$

and the output is

$$y(t) = \sin[2\pi f t + \hat{\phi}(t)]$$

leads to a signal z(t) of the form

$$z(t) = \frac{1}{2}\sin(\hat{\phi} - \phi) + \frac{1}{2}\sin(4\pi f t + \hat{\phi} + \phi)$$

The low pass filter (LPF) removes the doubled frequency component, giving the control voltage signal

$$v(t) \approx \frac{1}{2}\sin(\hat{\phi} - \phi) \approx \frac{1}{2}(\hat{\phi} - \phi)$$

The second approximation is true when the phase difference is small, or the loop is "tracking". The VCO performs the operation

$$\hat{\phi}(t) = -k \int_{-\infty}^{t} v(\tau) d\tau$$

The VCO therefore integrates the control voltage to compute what the phase of its output sinusoid should be. The term *k* is referred to as the *loop gain*. In the tracking mode, you can think of the whole loop as being a linear system, where the input and output *phase* are related by the transfer function

$$\frac{\hat{\Phi}(s)}{\Phi(s)} = \frac{kG(s)/s}{1 + kG(s)/s}$$

where G(s) is the response of the loop filter. Note that the factor of 0.5 has been lumped into the loop gain *k*. Often, a simple first order loop filter is employed with the response

$$G(s) = \frac{1 + \tau_2 s}{1 + \tau_1 s}$$

and this leads to a second order response for the linear PLL, and depending on the coefficients tau1 and tau2, the response is classified as overdamped, critically damped, or underdamped. A strongly overdamped system requires a long time to adapt to phase changes. An underdamped system has the characteristic of fast response, but a tendency to overshoot the target, leading to oscillations. The loop filter parameters can be derived from the desired overall loop response as

$$\tau_1 = \frac{k}{\omega_0^2}$$

$$\tau_2 = \frac{2D}{\omega_0} - \frac{1}{k}$$

where *D* is the damping factor and *w0* is the corner frequency. You can think of the corner frequency *w0* as controlling how fast the loop adapts to phase changes, and usually this frequency should be much lower (<0.1) than the frequency of the sinusoid or clock being tracked. The damping factor *D* is often chosen to be approximately 1, meaning critical damping. For some applications, it may be desired to have a slower response, and so *D* is made greater than 1.

Lookup Tables

For the PLL, we are going to need to access to sinusoidal functions, and in writing mathematical code in the past, you may be accustomed to doing things like

```
#include "math.h"

process()

{

    int n;

    float input, output;



    for (n=0; n<N_PERIOD; n++)

    {

        output = input*cos(2.0*pi*freq*n/sample_rate);



        /* More processing here */

    }

}
```

The problem with this kind of code on the DSP is that you want your code to be as efficient as possible in order for the algorithm to run in real-time. Also, DSP algorithms often use the same patterns (like the sine wave in this example) over and over. Calling trig functions like`cos()` computes the result using a polynomial approximation, requiring around 10 multiplies or so per result, which is not very efficient.

To make the code more efficient, we use a *lookup table*. Consider the following code:

```
#include "math.h"


float cos_table[N_PERIOD];


init()
{
    int n;


    for (n=0; n<N_PERIOD; n++)
    {
       cos_table[n] = cos(2.0*pi*freq*n/sample_rate);
    }
}


process()
{
    int n;
    float input, output;


    for (n=0; n<N_PERIOD; n++)
    {
        output = input*cos_table[n];


        /* More processing here */
    }
```

```
}
```

Now, we can just call the `init()` function once when our program initializes (like in a block init function). When we process signals in`process()`, we just use the lookup table, which can be much faster than recomputing all of those samples.

Discretization of the PLL

To map the continuous-time loop filter response H(s) to discrete time, we use the *bilinear transformation* or

$$s = \frac{2}{T}\frac{1 - z^{-1}}{1 + z^{-1}}$$

where *T* is the sample period and *z* is the discrete frequency from the Z transform. Making the appropriate substitutions, we derive the coefficients for our filter as

$$\begin{aligned} y[n] &= a_1 y[n-1] + b_0 x[n] + b_1 x[n-1] \\ a_1 &= -\frac{T - 2\tau_1}{T + 2\tau_1} \\ b_0 &= \frac{T + 2\tau_2}{T + 2\tau_1} \\ b_1 &= \frac{T - 2\tau_2}{T + 2\tau_1} \end{aligned}$$

In our analysis, we have used discrete frequency, which means that *T*=1.

The discretization of the VCO requires a little explanation. As explained above, we will use a lookup table rather than computing sin() and cos() directly. Also, for our discrete PLL, we will use a phase *accumulator* method, where instead of having an incrementing time variable *t*and a separate phase *phi*, we have a single real number, known as the accumulator, that keeps track of the current phase. At each time step, the contents of the accumulator tell us the current position in the sin() lookup table, where the lookup table holds just one sinusoidal cycle.

Note that in the lookup table example above, we only computed (and used) samples of sin() spaced by the sample period. This will not be the case for the PLL, since we will often need the “in between” samples, and the phase can change continuously to track the input. You can experiment with the size of the sine table, but I recommend that you use a large number of samples (say 1024) to be safe.

Although we can have our accumulator represent phase directly with period 2*pi, computations are more convenient if we just make the period 1.0. (Note: on fixed point architectures, you often make the period equal an integer multiple of the length of your sine table, so that everything can be done with integer arithmetic.)

Thus, on each time step, we do the following operation on our accumulator:

$$accum \leftarrow accum + f - \frac{k}{2\pi}v[n]$$

The first term is just the deterministic phase increase of our sinusoidal signal due to the onward advance of time. The second term is the additional phase shift due to the control signal present on the VCO. Note that we assume everything is normalized with respect to sample time (T=1), so that frequency is discrete (cycles/sample).

Since our sinusoidal period is 1.0, the accumulator needs to wrap when its value increments past 1 or drops below 0. This can be accomplished in either MATLAB or C (be sure to include “math.h”) with

```
accum = accum - floor(accum);
```

At each time step, the output of the VCO can then be computed in C as

```
s = sin_table[(int)((float)SIN_TABLE_SIZE*accum)];
```

Procedure

Loop Filter Design

Compute the filter coefficients for your first oder loop filter assuming the following specifications:

PARAMETER	DESCRIPTION	VALUE
f	Nominal ref. frequency	0.1
D	Damping factor	1.0

k	Loop gain	2.0
w0	Loop corner frequency	2*pi/100

These were chosen so that the response time of the loop is about 100 samples (10 sinusoidal cycles). For a sample rate of 8000 Hz, the center frequency is 800 Hz. Note that you should use 1.0 for *k* in computations due to the factor of 0.5 that was absorbed into *k* in the analysis.

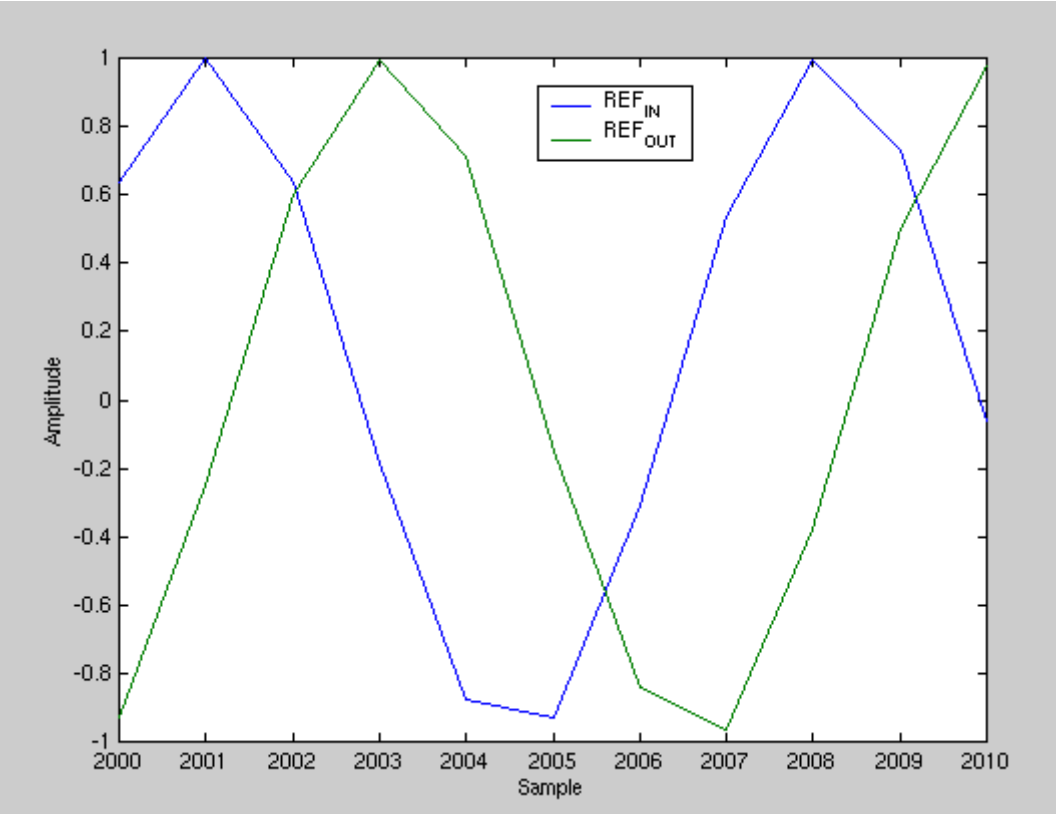
Simple MATLAB Implementation

Test the response of your loop filter in MATLAB and compare its response with the ideal frequency response. Refer back to the FIR filter lab for information on how to test a filter and plot its response.

Implement the complete PLL in MATLAB using an accumulator. Test its operation with a number of different input signals. The input signals should be sinusoids with frequencies that are within 10% of the nominal reference frequency.

To get you started, you can use the signal `ref_in` contained in the data files ref_800hz.mat and ref_stepf.mat. The first file gives you a constant 800 Hz sinusoid, which you should try first. In the second file, the signal steps through 4 different frequencies (see the variable`freq_in` for which frequency is active at which time). For each step to a new frequency, the PLL should undergo a transition and then eventually track.

To give you an idea of what “tracking” means graphically, the figure below shows the result of my PLL implementation at a point when tracking is achieved. You can see that the input and output frequency are identical, and that a constant 90 degree phase shift is maintained between the two. Plotting the two waveforms before or after this point, you will see the same relationship. If you see one waveform “slipping” relative to the other over time, your loop is not working!



MATLAB Implementation for Fixed Buffer Sizes

Now implement your PLL as a block in MATLAB, like you did in previous labs with the FIR filter and delay blocks. This will serve as a model for your C code that you will run on the DSP. Refer back to previous labs if you need help remembering how to store and use the state required by the PLL.

Note that although the PLL uses previous samples, you will not need a circular buffer for this. The PLL only needs to keep track of the accumulator and perhaps 2 old samples for the IIR filter, which can be done with just separate variables.

Test your buffer-oriented code and make sure the PLL still locks to your test signals.

MATLAB Implementation with Arbitrary Amplitude

In the development above, we assumed that the amplitude of the input signals was 1, which simplified the analysis. Also, the signals you generated to test your MATLAB implementation were also probably unit amplitude. If the amplitude of the input reference is not unity, this effectively changes the loop-gain of your filter, and may make the loop unable to track.

To solve this problem, you should change your PLL so that each block of samples is scaled to have approximately unit amplitude. Consider the pseudocode below:

```
amp = 0;
```

```
for sample_idx = 1:number_samples,

  amp = amp + abs(sample[sample_idx]);



  sample_scale = sample[sample_idx]/amp_est;



  ... Do processing here on sample_scale ...



end



amp_est = amp/number_samples/(2/pi);
```

In this code, we estimate the amplitude of each block of samples by just averaging the magnitude of the samples. That amplitude estimate is stored and then used to scale the *next* block of samples. Since the amplitude should not change drastically from one block to the next, this will make the amplitude of the `sample_scale` signal close to 1.

After changing your code to handle non-unit amplitude, scale the provided waveforms and make sure your PLL can still track them.

DSP Implementation

You will implement the C version of your PLL algorithm for the DSP in next week's lab.

Pre-Lab Exercise

You should have the following things prepared when you come to the first PLL lab:

Questions

Please provide *short* answers (say 1-3 sentences) for the following questions. In the pre-lab quiz we will ask to see your answers and may ask you to explain what they mean.

1. What are the two functions of a PLL in a communications system?
2. What are the key components (operations) used to implement a PLL?
3. When we write the second-order frequency response of the PLL, this expression relates the input and output _____ of the references.
4. What does the corner frequency of the PLL loop filter control?

5. How should the loop filter corner frequency compare to the input reference frequency?
6. What does it mean for a PLL to *track*?
7. What does it mean for the PLL loop to be overdamped or underdamped?
8. Why is an accumulator useful for a PLL implementation?
9. Why do we need to sample sin() in our lookup table more finely than at the normal sample rate of the system?
10. How do we keep our accumulator in the range [0,1]?
11. What is the point of storing and restoring the state of the PLL for each block?
12. How do we handle signals with arbitrary amplitude?

Testing of Matlab Code

Be able to explain how you tested your code and show plots of the resulting performance. Indicate any problems you observed with the operation of your PLL and how you intend to debug and fix them. Your PLL code might not work perfectly, but it should at least run without generating MATLAB errors. In the lab, we can help you debug and improve your design, but you should have made a reasonable attempt at the implementation.

Check-Off

Demonstrate to the TA that your PLL implementations work correctly. Your PLL should be able to handle changes in frequency of about plus or minus 10% of the nominal reference frequency.

Lab Write Up

Include the following items in your final write-up:

1. Answers to the questions asked in the pre-lab.
2. A paper design of your PLL, showing the important parameters that are needed for implementation
3. A printout of your final MATLAB implementation of the PLL
4. A plot showing the simulated performance of the final PLL, demonstrating that the PLL can adapt to abrupt changes in frequency/phase